7. Graphics Operations

This section provides an overview of the graphics drawing capabilities of the TMS34010. Topics in this section include:

7.1 Graphics Operations Overview

The TMS34010 instruction set provides several fundamental graphics drawing operations:

- The PIXBLT and FILL instructions manipulate two-dimensional arrays of pixels.
- The LINE instruction implements the fast inner loop of the Bresenham algorithm for drawing lines.
- The DRAV (draw and advance) instruction draws a pixel and increments the pixel address by a specified amount. This function supports the implementation of incremental algorithms for drawing circles, ellipses, arcs, and other curves.
- The PIXT (pixel transfer) instruction transfers individual pixels from one location to another.

The PIXBLT instruction plays an important role in rapidly drawing high-quality, bit-mapped text. In particular, the PIXBLT B,XY and PIXBLT B,L instructions expand character patterns stored as bit maps (at one bit per pixel) into color or gray-scale characters of 1, 2, 4, 8 or 16 bits per pixel. This allows character shape information to be stored independently of attributes such as color and intensity, providing greater storage efficiency.

The TMS34010 provides several methods for processing the values of the source and destination pixels before the result is written to the destination. These operations include:

- Boolean and arithmetic pixel processing operations for combining source pixels with destination pixels.
- A plane mask which specifies which bits within pixels can be altered during pixel operations.
- Transparency, an option which permits objects written onto the screen to have transparent regions through which the background is visible.

Pixel processing, plane masking and transparency can be used simultaneously. These operations on pixel values can be used in combination with any of the pixel drawing instructions listed above. The arithmetic operations are especially important in displays that use multiple bits per pixel to encode color or intensity information. For example, the MAX and MIN operations allow two objects with antialiased edges to be smoothly merged into a single image.

The TMS34010 has features such as automatic window checking to support windowed graphics environments. Three window-checking modes are provided:

- Clipping a figure to fit a rectangular window.
- Requesting an interrupt on an attempt to write to a pixel *outside* of a window.
- Requesting an interrupt on an attempt to write to a pixel *inside* of a window.

The last of these modes can be used to identify screen objects that are pointed to by a cursor. The window checking modes can be used with any of the pixel drawing instructions that use XY addressing. Window checking is optional and can be turned off.

The TMS34010 provides further support for windowed environments by rapidly detecting the following conditions:

- Whether a *point* lies inside or outside a rectangular window.
- Whether a *line* lies entirely inside or entirely outside a window.

Lines that lie entirely outside a window can be trivially rejected, meaning that they take no further processing time. These conditions are detected via the CPW (compare point to window) instruction, which takes only one machine state to compare the XY coordinates of a point to all four sides of a window.

Another operation that occurs frequently in windowed environments is calculating the region where two rectangles intersect. This is a feature available with the PIXBLT and FILL instructions. Based on the window-checking mode, one of two methods can be selected to calculate the region of intersection:

- The destination pixel array is preclipped to a rectangular window before the PixBlt or fill operation begins.
- The intersection of the destination pixel array with a rectangular window is calculated, but no pixels are transferred.

7.2 Pixel Block Transfers

The TMS34010 supports a powerful set of raster operations, known as *PixBlts* (pixel block transfers), that manipulate two-dimensional arrays of bits or pixels. A pixel array is defined by the following parameters:

- A starting address (by default, the address of the pixel with the lowest address in the array)
- A width *DX* (the number of pixels per row)
- A height *DY* (the number of rows of pixels)
- A pitch (the difference between the starting addresses of two successive rows)

A pixel array appears as a rectangular area on the screen. The array pitch is the same in this case as the pitch of the display. The default starting address is the address of the pixel in the upper left corner of the rectangle. (This assumes that the ORG, PBH, and PBV bits in the CONTROL register are all set to their default value of 0.)

Two operands must be specified for a PIXBLT instruction:

- A *source* pixel array
- A *destination* pixel array

The two arrays must have the same width and height, although they may have different pitches. Each pixel in the source array is combined with the corresponding pixel of the destination array. A Boolean or arithmetic *pixel processing operation* is selected and applied to the PIXBLT operation. The default pixel processing operation is *replace*. If replace is selected, source pixel values are simply copied into destination pixels.

Before executing a PIXBLT instruction, load the following parameters into the appropriate GSP internal registers:

DYDX Composed of two portions: DX, which specifies the width of the array, and DY, which specifies the height of the array.

PSIZE Pixel size (number of bits per pixel).

SADDR Starting address of source array (XY or linear address).

DADDR Starting address of destination array (XY or linear address).

SPTCH Source pitch, or difference in memory addresses of two vertically adjacent pixels in the source array.

DPTCH Destination pitch, or difference in memory addresses of two vertically adjacent pixels in the destination array.

If either the source or destination array is specified in XY format, the contents of the CONVSP and CONVDP registers will be used in instances in which the Y component of the starting address must be adjusted prior to the start of the PixBlt. The Y component may require adjustment, either to preclip the array or to select a starting pixel in one of the lower two corners of the array.

Pitches and starting addresses must be specified separately for the two arrays (source and destination). The width, height, and pixel size are common to both arrays. (During a binary expand operation, only the destination pixel size is specified; the source pixel size is assumed to be one bit.)

The starting address of a pixel array can be specified as a linear (memory) address or as an XY address. Window checking can be used only when the destination array is pointed to by an XY address.

On-screen objects may be defined as XY arrays but may be more efficiently stored as linear arrays in off-screen memory. An array specified in linear format can be transferred to an array specified in XY format (and vice versa) by means of the PIXBLT L,XY and PIXBLT XY,L instructions.

The FILL instruction fills a specified destination pixel array with the pixel value specified in the COLOR1 register. A fill operation can be thought of as a special type of PixBlt that does not use a source pixel array. The source pixel value used in pixel processing is the value in the COLOR1 register. The destination array of a FILL instruction can be specified in either XY or linear format.

7.2.1 Color-Expand Operation

The TMS34010 allows shape information to be stored separately from attributes such as color and intensity. A shape can be stored in compressed form as a bit map containing 1s and 0s. The color information is added as the shape is drawn to the screen; the 1s in the bit map are expanded to the specified Color 1 value, and the 0s are expanded to the Color 0 value. This saves a significant amount of memory when the pixel size in the display memory is two bits or more.

Two PIXBLT instructions, PIXBLT B,XY and PIXBLT B,L, provide the color-expand capability. The source array for either instruction is a bit map (one bit per pixel) stored off-screen in linear format for greater storage efficiency. The destination array can be specified in either XY or linear format. The pixel size for the destination array is governed by the value in the PSIZE register. The colors to which the 1s and 0s in the source array are expanded are specified in the COLOR1 and COLOR0 registers.

A primary benefit of the color-expand capability is the reduction in table area needed to store text fonts. Font bit maps are stored in compressed form at one bit per pixel. The color-expand operation adds color to a character shape at draw time, allowing color to be treated as an attribute separate from the shape of the character. The alternative would be to store the fonts in expanded form, which can be costly. The amount of table storage necessary to store red letters A–Z, blue letters A–Z, and so on, multiplied by the number of font styles needed for an application program, would be prohibitive. Furthermore, the color-expand operation is inherently faster than using pre-expanded fonts because far fewer bits of character shape information have to be read from the font table when a character is drawn to the screen.

Figure 7-1 shows the expansion of a bit map, one bit per pixel and four bits wide, into four 4-bit pixels (transforming 0-1-1-0 into yellow-red-red-yellow, for example). Before transferring the expanded source array to the destination array, any of the Boolean or arithmetic pixel processing operations can be applied.

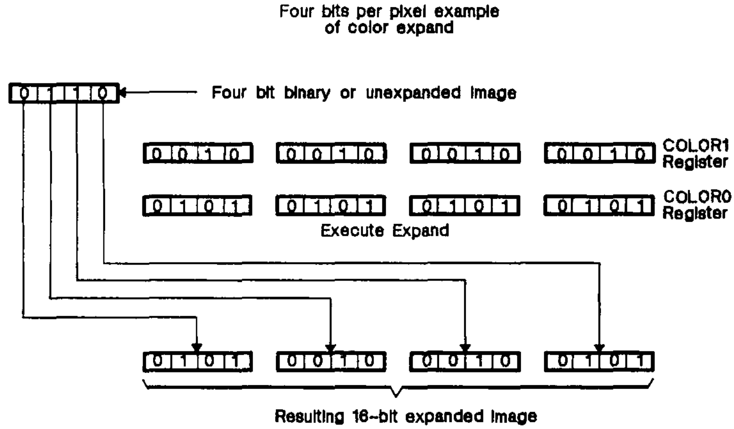


Figure 7-1. Color-Expand Operation

The expand function is also useful in applications that generate shapes or patterns dynamically. During the first stage of this process, a compressed image is constructed in an off-screen buffer area at one bit per pixel. The image is built up of geometric objects such as rectangles, circles or polygons. Patterns can also be added. When complete, the compressed image is color-expanded onto the screen. This method defers the application of color and intensity attributes until the final stage.

Combining color expand with the replace-with-transparency operation yields a new operation that is particularly useful in drawing overlapping or kerned text. The color value used to replace the 0s in the source array is selected by the programmer as all 0s, which is the transparency code. The GSP defers the check for transparency until after the color-expand operation has been performed. As the color-expand operation is performed, the 0s in the source array are expanded to all 0s. Only the pixels in the destination array that correspond to nontransparent pixels in the resulting source array are replaced.

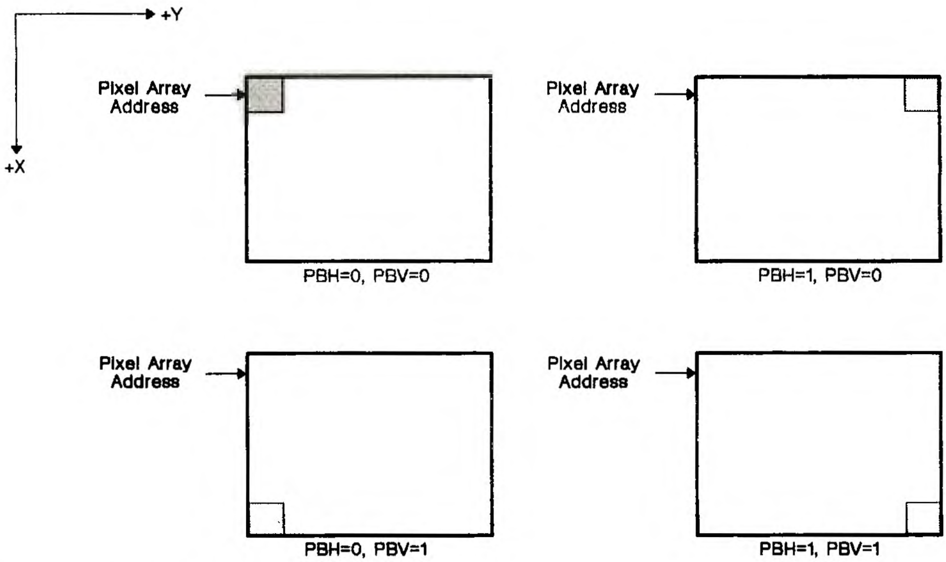
The PIXBLT B,XY and PIXBLT B,L instructions can be used in conjunction with pixel processing, transparency and plane masking. Source pixels are expanded before being processed. Window checking can be used with PIXBLT B,XY.

7.2.2 Starting Corner Selection

The default starting address of a pixel array is the lowest pixel address in the array. When an array is displayed on the screen, as shown in Figure 7-2 *a*, the starting address is the address of the pixel in the upper left corner of the array. (The XY origin is located in its default position at the upper left corner of the screen.) During a PixBlt operation, this pixel is processed first. The PixBlt processes pixels from left to right within each row, beginning at the top row and moving toward the bottom row. The pixel at the lower right corner of the array is processed last.

Certain PixBlt operations allow any of the other three corners to be used as the starting location. This may be necessary, for instance, if the source and destination arrays overlap. The sequence in which pixels are moved when the arrays overlap should be controlled so as to not overwrite the pixels in the source array before they are written to the destination array.

Figure 7-2 shows how the PBV and PBH bits in the CONTROL register determine the starting corner for the PixBlt operation. The starting corner is indicated for each of four cases. PBH selects movement in the X direction, from left to right or right to left. PBV selects movement in the Y direction, from top to bottom or bottom to top.



Note: Starting corners are shaded.

Figure 7-2. Starting Corner Selection

PBH=0 The PixBlt processes pixels from left to right; that is, in the direction of *increasing* X.

PBH=1 The PixBlt processes pixels from right to left; that is, in the direction of *decreasing* X.

PBV=0 The PixBlt processes rows from top to bottom; that is, in the direction of *increasing* Y.

PBV=1 The PixBlt processes rows from bottom to top; that is, in the direction of *decreasing* Y.

All the pixels in one row are processed before moving to the next row.

When one or both of the arrays is specified in XY format, the GSP automatically calculates the actual starting address (specified by PBH and PBV) from the default starting address (that is, the lowest pixel address in the array) and the width and height of the array. Automatic starting address adjustment is available with the following instructions:

- PIXBLT L,XY
- PIXBLT XY,L
- PIXBLT XY,XY

The programmer supplies the *default* starting addresses for these PixBlts in the SADDR and DADDR registers. During the course of instruction execution, SADDR and DADDR are automatically adjusted to the address of the corner selected by PBH and PBV.

When *both arrays are specified in linear format*, the starting addresses of the appropriate corner pixels must be provided by the programmer. The PIXBLT L,L instruction allows any of the four corners to be used as the starting location, but in this case the programmer must adjust the addresses in SADDR and DADDR to the corner selected by PBH and PBV.

7.2.3 Interrupting PixBlts and Fills

PIXBLT and FILL are interruptible instructions. An interrupt can occur during execution of one of these instructions; when interrupt processing is completed, execution of the PIXBLT or FILL resumes at the point at which the interruption occurred.

The execution time of a PIXBLT or FILL instruction depends on the specified pixel array size. In order to prevent high-priority interrupts from being delayed until completion of PixBlts and fills of large arrays, the PIXBLT and FILL instructions check for interrupts at regular intervals during their execution.

When a PIXBLT or FILL instruction is interrupted the PBX (PixBlt executing) status bit is set to 1. This records the fact that the interrupt occurred during a pixel array operation. The PC and the ST are pushed onto the stack, and control is transferred to the appropriate interrupt service routine. At the end of the interrupt service routine, an RETI (return from interrupt) instruction is executed to return control to the interrupted program. The RETI instruction pops the ST and PC from the stack. When the PBX bit is detected, execution of the interrupted PIXBLT or FILL instruction resumes.

At the time of the interrupt, the state of the PIXBLT or FILL instruction is saved in certain B-file registers. The source and destination address registers contain intermediate values. The source and destination pitches may also contain intermediate values, depending on the instruction. The SADDR, SPTCH, DADDR, DPTCH registers and registers B10-B14 (as well as the original set of implied operands) contain the information necessary to resume the instruction upon return from an interrupt.

If the interrupt routine uses any of these registers, they should be saved on the stack and restored when interrupt processing is complete. By following this procedure, PIXBLT or FILL instructions can be safely executed within interrupt service routines.

> **Note:**
>
> The PBX bit is not set to 1 when a PIXBLT or FILL instruction is aborted due to a window violation.

7.3 Pixel Transfers

The TMS34010 uses the PIXT (pixel transfer) instructions to transfer individual pixels from one location to another. The following pixel transfers can be performed:

- From an A- or B-file register to memory,
- From memory to an A- or B-file register,

  or
- From one memory location to another.

The address of a pixel in memory can be specified in XY or linear format. Linear addresses must be pixel aligned.

The pixel size for all PIXTs is specified by the value in the PSIZE register. Pixel sizes are restricted to 1, 2, 4, 8, or 16 bits to facilitate XY address computations, window checking, transparency, and arithmetic pixel processing.

The PIXT instruction can be used in conjunction with window checking, Boolean or arithmetic pixel processing, plane masking, and transparency.

7.4 Incremental Algorithm Support

The TMS34010 supports incremental drawing algorithms via its DRAV (draw and advance) and LINE instructions. The DRAV instruction is used primarily in the construction of algorithms for incrementally drawing circles, ellipses, arcs, and other curves. The DRAV instruction can also be used in the inner loop of algorithms for drawing straight lines incrementally. Lines, however, are treated as a special case by the TMS34010 in order to achieve even faster drawing rates. A separate instruction, LINE, implements the entire inner loop of the Bresenham algorithm for drawing lines.

The DRAV (draw and advance) instruction draws a pixel to a location pointed to by a register; the pointer register is then incremented to point to the next pixel. The pointer is specified as an XY address. The X and Y portions of the address are incremented independently, but in parallel. The value written to the destination pixel in memory is taken from the COLOR1 register.

The DRAV instruction is embedded in the inner loop of an incremental algorithm to speed up its execution. As an incremental algorithm plots each pixel on a curve, it also determines where the next pixel will be drawn. The next pixel is typically one of the eight pixels immediately surrounding the pixel just plotted on the screen. Advancing in this manner, the algorithm tracks the curve from one end to the other.

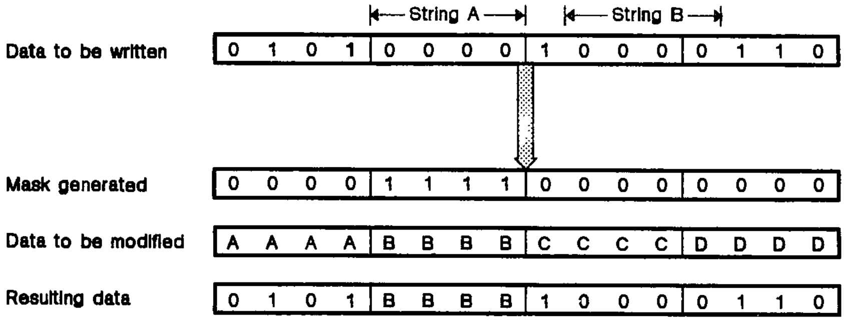
The DRAV and LINE instructions may be used in conjunction with Boolean or arithmetic pixel processing operations, window checking, plane masking and transparency.

7.5 Transparency

When a PixBlt is used to draw an object to the screen, some of the pixels in the rectangular pixel array that contains the object may not be part of the object itself. *Transparency* is a mechanism that allows surrounding pixels in the array to be specified as invisible. This is useful for ensuring that only the object, and not the rectangle surrounding it, is written to the screen.

Transparency is enabled by setting the T bit in the CONTROL register to 1, or disabled by setting the T bit to 0. When enabled, a pixel that has a value of 0 is considered transparent, and will not overwrite a destination pixel. *Transparency detection is applied not to the source pixel values, but to the pixel values resulting from plane masking and pixel processing.* When an operation performed on a pair of source and destination pixels yields a 0 result, the GSP detects this and prevents the destination pixel from being altered. In the case of pixel processing operations such as AND, MIN, and replace, a source pixel value of 0 ensures that the result of the operation will be a transparent pixel.

Figure 7-3 illustrates how transparency works in the GSP. Assuming four bits per pixel, the hardware must detect strings of 0s of length four falling between pixel boundaries. While bit strings ***A*** and ***B*** are both of pixel length, only string ***A*** is detected as transparent. String ***B*** crosses the pixel boundary. The memory interface logic generates an internal mask to govern which bits are modified during a write cycle. This mask contains 1s in the bits corresponding to the transparent pixel. Only destination bits corresponding to 0s in the mask will be modified.



Note: This example assumes four bits per pixel.

Figure 7-3. Transparency

Figure 7-7 (page 7-17) and Figure 7-8 (page 7-19) illustrate several pixel processing operations. Figure 7-8 *h* shows an example of a replace operation performed with transparency enabled. The pixels surrounding the letter **A** pattern in the source array are transparent (all 0s). Compare Figure 7-8 *h* with Figure 7-7 *d*; this replace-with-transparency operation is analogous to the logical OR operation in a one-bit-per-pixel display.

Transparency can be used with any instruction that writes to pixels, including the PIXBLT, FILL, DRAV, LINE, and PIXT instructions. Transparency does not affect writes to non-pixel data.

7.6 Plane Masking

The plane mask is a hardware mechanism for protecting specified bits within pixels. Mask-protected pixels will not be modified during graphics instructions. The plane mask allows the bits within pixels to be manipulated as though the display memory were organized into *bit planes* (or *color planes*) that can selectively be protected from modification. The number of planes equals the number of bits per pixel.

Consider an example in which the pixel size is four bits. The bits within each pixel are numbered 0–3, and belong to planes 0–3, respectively. All the bits numbered 0 in all the pixels form plane 0, all the bits numbered 1 in all the pixels form plane 1, and so on.

The plane mask allows one or more planes to be manipulated independently of the other planes. Given four planes of display memory, for example, three of the planes can be dedicated to eight-color graphics, while the fourth plane can be used to overlay text in a single color. The plane mask can be set so that the text plane can be modified without affecting the graphics planes, and vice versa.

The PMASK register contains the plane mask. Each bit in the plane mask corresponds to a bit position in a pixel. The 1s in the mask designate pixel bits that are protected, while 0s in the mask designate pixel bits that can be modified. Those pixel bits that are protected by the plane mask are always read as 0s during read cycles, and are protected from alteration during write cycles. While no single control bit enables or disables plane masking, it is effectively disabled by setting PMASK to all 0s; this is the default condition following reset.

In principal, the number of bits in the plane mask is the same as the pixel size. However, the mask for a single pixel must be replicated to fill the entire 16-bit PMASK register. For example, if the pixel size is four bits, the 4-bit mask is replicated four times within PMASK; in bits 0–3, 4–7, 8–11, and 12–15. These four copies of the mask are applied to the four pixels in a word written to or read from memory. A 16-bit PMASK value for pixels of 1, 2, 8, or 16 bits is constructed similarly by replicating the mask 16, 8, 2, or 1 times, respectively.

The plane mask affects only pixel accesses performed during execution of the PIXBLT, FILL, PIXT, DRAV, and LINE instructions. Data accesses by non-graphics instructions are not affected.

The following list summarizes operation of the PMASK register during pixel reads and writes:

- Pixel Read:

  The **0s** in PMASK correspond to unprotected bits in the source pixel that are seen by the GSP to contain the actual values read from memory.

  The **1s** in PMASK correspond to protected bits in the source pixel that are seen as 0s by the GSP, regardless of the values read from memory.

- **Pixel Write:**

  The **0s** in PMASK specify those bits in the destination pixel in memory which may be altered.

  The **1s** in PMASK specify protected bits in the destination pixel which cannot be altered.

When a pixel is being transferred from a source to a destination location, plane masking is applied to the values read from the source and destination before pixel processing is applied. As the operands are read from memory, the bits protected by the plane mask are replaced with 0s *before* the specified Boolean or arithmetic pixel processing operation is performed. Transparency detection is performed on the result of this operation. When the result is written back to the destination, those bits of the destination that are protected by the plane mask are not modified.

Source pixels that originate from registers are not affected by the plane mask, and undergo pixel processing in unmodified form. The FILL, DRAV, LINE, PIXT Rs,*Rd, and PIXT Rs,*Rd.XY instructions obtain their source pixels from registers.

Figure 7-4 shows how special hardware in the local memory interface of the TMS34010 applies the plane mask to pixel data during a read cycle. The pixel size for this example is eight bits per pixel. This could represent the execution of a PIXT *Rs.XY,Rd instruction, for instance.

Move this pixel into a GSP register

	15(MSB)							8	7							0(LSB)
(a) Original data in memory (2 pixels)	A	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B
(b) Plane mask (PMASK)	1	1	0	0	1	0	0	0	1	1	0	0	1	0	0	0
(c) Data read into GSP register	0	0	0	0	0	0	0	0	0	0	B	B	0	B	B	B

Notes: 1. This example assumes eight bits per pixel.
2. The pixel moved into the GSP register is left justified. All register bits to the left of the pixel are zero filled.

Figure 7-4. Read Cycle With Plane Masking

- Figure 7-4 *a* shows the 16-bit word containing the pixel as it is read from memory.
- The word is ANDed with the inverse of the plane mask shown in *b*.
- The result in Figure 7-4 *c* shows that the bits within the data word that correspond to 1s in the mask have been set to 0s.

After plane masking, the designated pixel is loaded into the eight LSBs of the 32-bit destination register, and the 24 MSBs of the register are filled with 0s.

Figure 7-5 shows the effect of combining plane masking with pixel transparency. Again, the performance of the special hardware in the local memory interface controller is demonstrated. The example shows the transfer of two pixels during the course of a PixBlt operation with transparency enabled, the pixel size set at eight bits, and the *replace* pixel processing operation. The inverse of PMASK is ANDed with the source data, and transparency detection is applied to the resulting entire pixel. In other words, the result is used to control the write in the manner described in the previous discussion of pixel transparency. Since the three LSBs of the source pixel in bits 8–15 are 0s, and the rest of the pixel is masked off, the entire source pixel is interpreted as transparent. The memory interface logic generates an internal mask to govern which bits are modified during a write cycle. This mask contains 0s in the bits corresponding to the transparent pixel.

	15(MSB)							8	7							0(LSB)
(a) Original data in memory (2 pixels)	A	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B
(b) Source data in memory (to be moved)	Y	Y	Y	Y	Y	0	0	0	Z	Z	Z	Z	Z	Z	Z	Z
(c) Plane mask (PMASK)	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0
(d) Mask source data for transparency detection (SRC • $\overline{\text{PMASK}}$)	0	0	0	0	0	0	0	0	0	0	0	0	0	Z	Z	Z
(e) Transparency mask	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
(f) Combined mask ($\overline{\text{PMASK}}$ • transparency mask)	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
(g) Resulting memory data after write cycle (Combined Mask • SRC DATA + $\overline{\text{Combined Mask}}$ • DST DATA)	A	A	A	A	A	A	A	A	B	B	B	B	B	Z	Z	Z

Note: This example assumes eight bits per pixel.

Figure 7-5. Write Cycle With Transparency and Plane Masking

- Figure 7-5 *a* shows the original data at the destination location in memory.
- The source data are shown in *b*.
- The source data are ANDed with the inverse of the plane mask shown in *c*.
- Figure 7-5 *d* shows the intermediate result produced by *c*.
- This result is used to generate the transparency mask in *e*, which is ANDed with the inverse of the plane mask in *c* to produce the composite mask shown in *f*.
- The result in *g* is produced by replacing with the source only those bits of the destination corresponding to 1s in the composite mask in *f*.

7.7 Pixel Processing

Source and destination pixel values can be combined according to the *pixel processing* operation (or raster operation) selected. The TMS34010's pixel processing operations include 16 Boolean and 6 arithmetic operations. The Booleans are performed in bitwise fashion on operand pixels of 1, 2, 4, 8, or 16 bits. The arithmetic operations treat operand pixels of 4, 8, or 16 bits as 2's complement integers.

When a pixel is read from its source location, it is logically or arithmetically combined with the corresponding destination pixel according to the pixel processing option selected, and the result is written to the destination pixel. The pixel processing operation is selected by the PPOP field in the CONTROL register. Table 7-1 and Table 7-2 list the 22 PPOP codes and their meanings.

Table 7-1. Boolean Pixel Processing Options

PPOP Field	Operation
00000	Source → Destination
00001	Source AND Destination → Destination
00010	Source AND ~Destination → Destination
00011	0s → Destination
00100	Source OR ~Destination → Destination
00101	Source XNOR Destination → Destination
00110	~Destination → Destination
00111	Source NOR Destination → Destination
01000	Source OR Destination → Destination
01001	Destination → Destination
01010	Source XOR Destination → Destination
01011	~Source AND Destination → Destination
01100	1s → Destination
01101	~Source OR Destination → Destination
01110	Source NAND Destination → Destination
01111	~Source → Destination

Table 7-2. Arithmetic (or Color) Pixel Processing Options

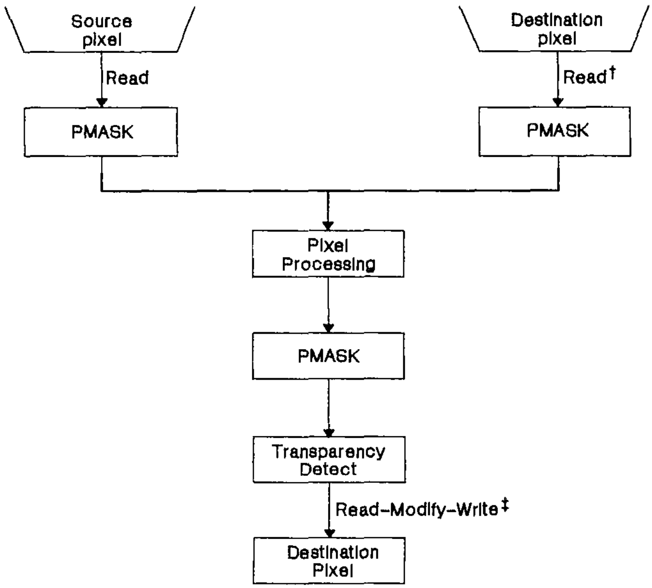
PPOP Field	Operation
10000	Source + Destination → Destination
10001	ADDS(Source, Destination) → Destination
10010	Destination - Source → Destination
10011	SUBS(Source, Destination) → Destination
10100	MAX(Source, Destination) → Destination
10101	MIN(Source, Destination) → Destination
10110-11111	Reserved

In Table 7-2, pixel processing codes 10000 and 10010 correspond to standard 2's complement addition and subtraction. A result that overflows the specified pixel size causes the pixel value to wrap around within its 4, 8, or 16-bit range. Carry bits are, however, prevented from propagating to adjacent pixels.

The ADDS (add with saturation) and SUBS (subtract with saturation) operations shown in Table 7-2 produce results identical to those of standard addition or subtraction, except when arithmetic overflow occurs. When the ADDS operation would produce an overflow result, the result is replaced with all 1s. When the SUBS operation would produce an underflow result, the result is replaced with all 0s.

The MAX operation shown in Table 7-2 compares the source and destination pixels and then writes the greater value to the destination location. The MIN operation is similar, but writes the lesser value to the destination.

Figure 7-6 depicts the interaction of pixel processing with other graphics operations when a source pixel is transferred to a destination pixel. Note that this is a general description; some of these operations do not occur if they are not selected. Pixels are first read from memory and modified by the plane mask. Pixel processing is then performed on the modified pixel values. The plane mask is applied to the result. Bits which are 1s in the PMASK produce 0 bits in the result of this process. Thus, some processed pixels may become transparent as the result of plane masking. Next, transparency detection is applied to the data, and finally, a read-modify-write operation is invoked.



† Only necessary if *replace* is not selected.
‡ Only necessary when plane masking or transparency is active and the pixel size is not 16, or when the data is not word-aligned.

Figure 7-6. Graphics Operations Interaction

7.8 Boolean Processing Examples

Figure 7-7 illustrates the effects of five commonly used Boolean operations when applied to one-bit pixels. Black regions contain 0s, and white regions contain 1s. Figure 7-7 *a* and *b* show the original source and destination arrays. The source operand in *a* is the letter **A**, and the destination in *b* is a calligraphic-style **X**.

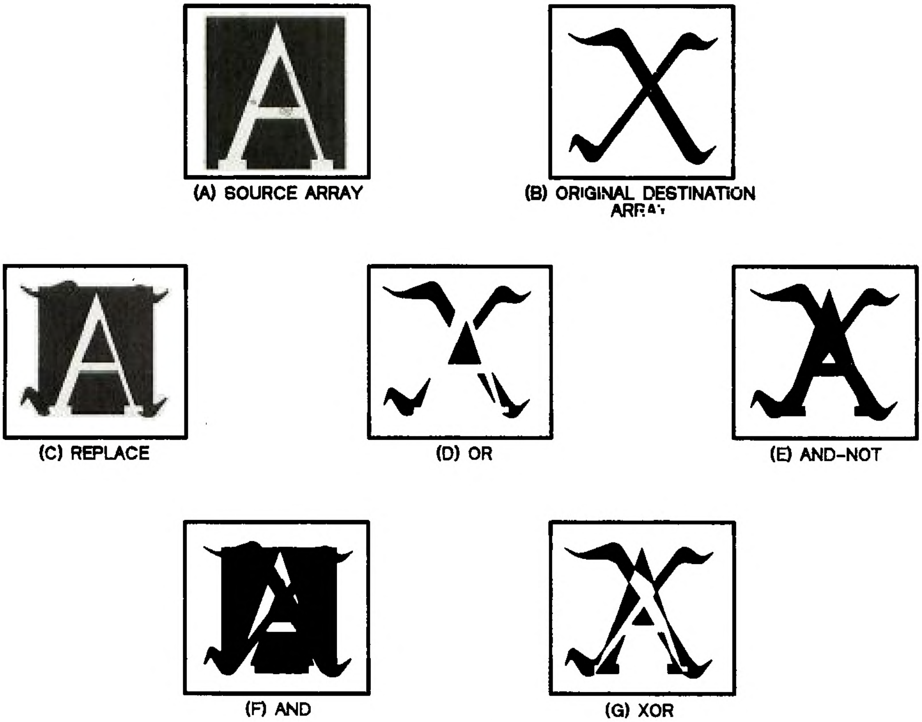


Figure 7-7. Examples of Operations on Single-Bit Pixels

7.8.1 Replace Destination with Source

A simple replacement operation overwrites the pixels of the destination array with those of the source. Figure 7-7 *c* shows the letter **A** written over the center portion of a larger **X** using the replace operation. The rectangular region around the letter **A** obscures a portion of the **X** lying outside the **A** pattern. Other operations allow only those pixels corresponding to the **A** pattern within the rectangle to be replaced, permitting the background pattern to show through. These are the logical OR and logical AND-NOT (NOT source AND destination) operations. The replace-with-transparency operation performs similarly in color systems.

7.8.2 Logical OR of Source with Destination

Figure 7-7 *d* illustrates the use of the logical OR operation during a PixBlt. For a one-bit-per-pixel display, the OR function leaves the destination pixels unaltered in locations corresponding to 0s in the source pixel array. Destination pixels in positions corresponding to 1s in the source are forced to 1s.

7.8.3 Logical AND of NOT Source with Destination

Logically ANDing the negated source with the destination is complementary to the logical OR operation. Destination pixels corresponding to 1s in the source array remain unaltered, but those corresponding to 0s in the source are forced to 0s. Figure 7-7 *e* is an example of the AND-NOT PixBlt operation (notice the negative image of the letter **A**). For comparison, Figure 7-7 *f* shows the result of simply ANDing the source and destination.

7.8.4 Exclusive OR of Source with Destination

The XOR operation is useful in making patterns stand out on a screen in instances where it is not known in advance whether the background will be 1s or 0s. At every point at which the source array contains a pixel value of 1, the corresponding pixel of the destination array is flipped - a 1 is converted to a 0, and vice versa. XOR is a reversible operation; by XORing the same source to the same destination twice, the original destination is restored. These properties make the XOR operation useful for placing and removing temporary objects such as cursors, and in "rubberbanding" lines. As seen in the example of Figure 7-7 *g*, however, the object may be difficult to see if both the source and destination arrays contain intricate shapes.

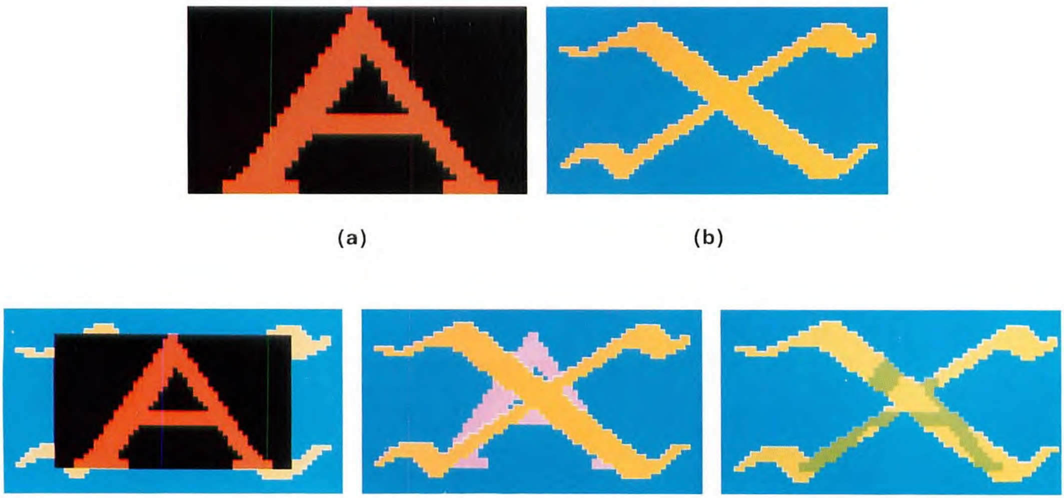
7.9 Multiple-Bit Pixel Operations

The Boolean operations described in Section 7.8 are sufficient for single-bit pixel operations, but they may be inappropriate for multiple-bit pixel operations, especially when color is involved. For example, the result of a logical OR operation on a black-and-white (one bit per pixel) display is easily predicted – logically ORing black and white yields white. However, the intuitive meaning of this operation is less clear when it is applied to multiple-bit pixels; what effect should be expected when the color red is ORed with blue?

7.9.1 Examples of Boolean Operations

Boolean operations can be applied to multiple-bit pixels by combining the corresponding bits of each pair of source and destination pixels on a bit-by-bit basis according to the specified Boolean operation.

Figure 7-8 illustrates Boolean operations on multiple-bit pixels. Figure 7-8 *a* illustrates the source array. It contains a red letter **A** which has the value 8 (1000_2); the black background pixels have the value 0 (0000_2). Figure 7-8 *b* shows the destination array, a yellow **X** which has the value 12 (1100_2); the pixels in the blue rectangle have the value 2 (0010_2). Figure 7-8 *c* through *g* show the effects of combining the source and destination arrays using the replace, logical OR, OR-NOT, AND and XOR PixBlt operations. Compare these to Figure 7-7 (page 7-17). Figure 7-8 *i* through *n* are discussed in Section 7.9.1.1 through Section 7.9.1.4.



(a) (b)

(c) (d) (e)

Figure 7-8. Examples of Boolean Operations

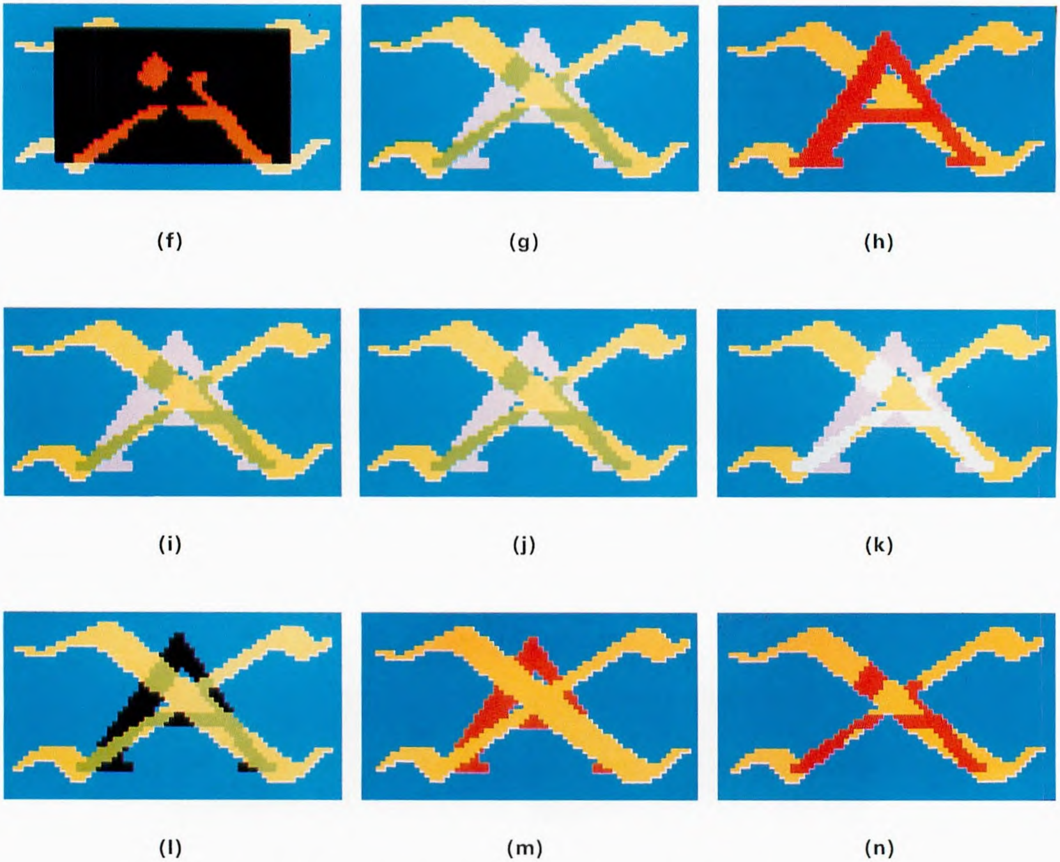


Figure 7-8. Examples of Boolean Operations (Concluded)

7.9.1.1 Figure 7-8 i and j - Simple Addition and Subtraction

Figure 7-8 *i* shows the result of adding the source and destination arrays. Simple binary 2's complement addition is used. When the sum of the two pixels exceeds the maximum pixel value, the result overflows.

Figure 7-8 *j* shows the result of subtracting the source array from the destination array. Underflow occurs for those pixels whose calculated difference is negative.

Simple addition and subtraction are complementary operations. They are reversible operations in the same sense as the XOR operation - by adding a source to a destination, and then subtracting the same source, the original destination is recovered.

7.9.1.2 Figure 7-8 k and l - Add and Subtract with Saturate

The add and subtract operations described in Section 7.9.1.1 are binary 2's complement operations which allow overflow and underflow. An add-with-saturate operation can be defined that stops the result at the maximum value rather than allowing it to overflow. For example, with four bits per pixel, adding 0010_2 to 1110_2 produces 1111_2. Similarly, a subtract-with-saturate operation can be defined that stops the result at 0 rather than allowing it to underflow.

Figure 7-8 *k* and *l* illustrate examples of add and subtract with saturate. In these examples, the pixel size is four bits. By dedicating a different color to each value, the effects of each PixBlt operation become more visible. This method may present problems, however. For example, adding red to blue may not produce a meaningful result.

An alternate method uses the 16 values 0 to 15 to represent increasing intensities of a single color. Then the addition and subtraction operations would have obvious meaning - they would increase and decrease the intensity by known amounts. Developing this idea further, at 12 bits per pixel, four bits of intensity could be dedicated to each of the three color components, red, green and blue. Arithmetic operations could then be performed on the corresponding components of each pair of source and destination pixels. These results would also have obvious meanings, and would not be limited to intensities of a single color, as is the case with four bits per pixel.

Figure 7-9 (page 7-22) presents examples in which the pixel values represent intensities of a single color.

7.9.1.3 Figure 7-8 m - Maximum

Figure 7-8 *m* illustrates the results of the MAX operation on the source and destination arrays. MAX compares two pixel values and replaces the destination pixel with the larger value. In some respects, MAX is the arithmetic equivalent of the Boolean OR function (compare Figure 7-8 *m* with Figure 7-7 *b*). The use of MAX in gray-scale and color displays is similar to that of OR in simple black and white.

If the most-significant bits in each pixel are assigned to represent object priority (whether an object appears in front of or behind another object), the MAX operation can be used to replace only those pixels of the destination array whose priorities are lower than those of the corresponding pixels in the source array. This allows an object to be drawn to the screen so that it appears either in front of or behind other objects previously drawn. In Figure 7-8 *m* the red **A** has a numerical value that is greater than that of the blue background, but less than that of the **X**.

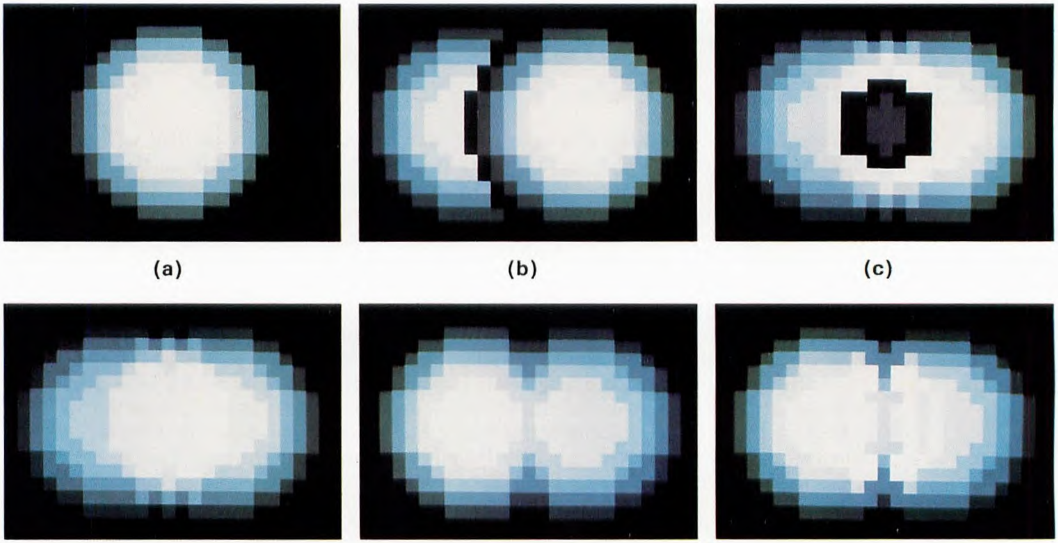
The MAX function is also useful for smoothly combining two antialiased objects that overlap.

7.9.1.4 Figure 7-8 n - Minimum

Figure 7-8 *n* illustrates the results of the MIN operation on the source and destination arrays. MIN compares two pixel values and replaces the destination pixel with the smaller value. MIN is similar to the Boolean AND function. MIN can be used with priority-encoded pixel values, similar to MAX, but the effect is reversed. In Figure 7-8 *n*, the priorities of the two objects are reversed from that of the MAX example shown in Figure 7-8 *m*. The MIN operation also has uses similar to those of MAX in smoothly combining antialiased objects that overlap.

7.9.2 Operations On Pixel Intensity

Figure 7-9 illustrates the visual effects of various PixBlt operations on two intersecting disks. In these examples, each pixel is a four-bit value representing an intensity from 0 (black) to 15 (white). Before the PixBlt operation, only a single disk resides on the screen, as shown in Figure 7-9 *a*. The intensity of the disk is greatest at the center (where the value is 12), and gradually falls off as the distance from the center increases. Figure 7-9 *b* through *f* show the effects of combining a second, identical disk with the first. Figure 7-9 *b* through *e* are produced using arithmetic operations; *f* is the result of a logical OR of the source and destination. These operations are discussed in Section 7.9.2.1 through Section 7.9.2.4.



(a) (b) (c) (d) (e) (f)

Figure 7-9. Examples of Operations on Pixel Intensity

The gradual change in intensity at the edge of the disk in Figure 7-9 *a* is similar to the result produced by certain antialiasing techniques whose purpose is to reduce jagged-edge effects. A text font might be stored in antialiased form, for example, to give the text a smoother appearance. When two characters from the font table are PixBlt'd to adjacent positions on the screen, they may overlap slightly. The particular arithmetic or Boolean operation selected for the PixBlt determines the way in which the antialiased edges of the characters are combined within regions of overlap.

7.9.2.1 Figure 7-9 b - Replace with Transparency

In Figure 7-9 *b*, a second disk is PixBlt'd into a position near the first disk. A replace-with-transparency operation is performed. Those pixels of the first disk that lie within the rectangular region containing the second disk, but are not part of the second disk, remain intact. The visual effect is that the second disk (at the right) appears to lie in front of the original disk (at the left). However, assuming that the gradual change in intensity at the perimeter of the disks is done for the purpose of antialiasing, the sharp edge that results where the second disk covers the first defeats this purpose. In other applications, this sharp edge may be desirable; for example, it might be used to make a text character or a cursor stand out from the background. The replace-with-transparency operation also supports object priority by writing objects to the screen in ascending order of priority.

7.9.2.2 Figure 7-9 c - Add with Overflow and Subtract with Underflow

In Figure 7-9 *c*, a second disk is PixBlt'd into an area overlapping the first disk, using an add-with-overflow operation. In this example, when 1 is added to an intensity of 15, the sum is truncated to four bits to produce the result 0. The effect of arithmetic overflow is visible at the intersection of the two disks as discontinuities in intensity.

This effect is useful for making objects stand out against a cluttered background. Add with overflow has an additional benefit – the object can be removed by subtracting (with underflow) the object image from the screen.

7.9.2.3 Figure 7-9 d - Add and Subtract with Saturation

In Figure 7-9 *d*, the original disk is on the left. A second disk is PixBlt'd into a region overlapping the original disk, using an add-with-saturate operation. Whenever the sum of two pixels exceeds the maximum intensity value, which is 15 for this example, the sum is replaced with 15. The bright region that occurs where the two disks intersect is produced when the corresponding pixels of the two disks are added in this manner. Subtract-with-saturate is the complementary operation; when the difference of the two pixel values is negative, the sum is replaced by the minimum intensity value, 0.

The add-with-saturate operation shown in Figure 7-9 *d* approximates the effect of two light beams striking the same surface; the surface is brightest in the area in which the two beams overlap.

These operations can be used to achieve an effect similar to that of an airbrush in painting. Consider a display system that represents each *pixel* as 12 bits, and dedicates four bits each to represent the intensities of the three color components, red, green, and blue. This method permits the intensity of each component to be directly manipulated. With each pass of the simulated airbrush over the same area of the screen, the color changes gradually toward the color of the paint in the airbrush. For example, assume that the paint is yellow (a mixture of red and green). Each time a pixel is touched by the airbrush, the intensity of the red and green components is increased by 1, and the intensity of the blue component is decreased by 1. With each sweep of the airbrush, the affected area of the screen turns more yellow until the red and green components reach the maximum intensity value (and are not allowed to overflow), and the blue component reaches 0 (and is not allowed to underflow).

7.9.2.4 Figure 7-9 e - MAX and MIN Operations

In Figure 7-9 *e*, the original disk is on the left. A second disk is PixBlt'd into the rectangular region to its right using the MAX operation. In the region in which the disks overlap, each pair of corresponding pixels from the two disks is compared and the greater value is selected. This produces a relatively smooth blending of the two disks. Unlike add with saturate, the MAX function does not generate a "hot spot" where two objects intersect.

The visual effect achieved using the MAX operation is desirable in an application, for instance, in which white antialiased lines are constructed on top of each other over a black background. MAX also smooths out places in which the lines are overlapped by antialiased text. MAX is successful in maintaining two visually distinct antialiased objects, while the add-with-saturate tends to run them together.

MIN, which is complementary to MAX, can be used similarly to smooth the appearance of intersecting black antialiased lines and text on a white background.

The MAX and MIN operations are particularly useful in color applications in which the number of bits per color gun is small (eight bits or less). Other operators could also be used to smooth the transition between the two overlapping antialiased objects in Figure 7-9 *e*, but any additional accuracy attained by using a more complex smoothing function would probably be lost in truncating the result to the resolution of the integer used to represent the intensity at each point.

7.10 Window Checking

The TMS34010's hardware *window clipping* confines graphics drawing operations to a specified rectangular window in the XY address space. Other window checking modes cause an interrupt to be requested on a window *hit* or a window *miss*.

Window checking affects only pixel writes performed by the following graphics instructions:

- PIXBLT
- FILL
- LINE
- DRAV
- PIXT

Data writes by non-graphics instructions are not affected.

A *window* is a rectangular region of display memory specified in terms of the XY coordinates of the pixels in its two extreme corners (minimum X and Y, and maximum X and Y). The corner pixels are considered to lie within the window. Window checking is available only in conjunction with XY addressing; it is not available with linear addressing. Specifically, the destination pixel address must be an XY address.

One of four window checking modes is selected by the value loaded into the W field of the CONTROL register:

W=0: *Window checking disabled.* No window checking is performed.

W=1: *Window hit detection.* Request interrupt on attempt to write *inside* window.

W=2: *Window miss detection.* Request interrupt on attempt to write *outside* window.

W=3: *Window clipping.* Clip all pixel writes to window.

When window checking is enabled (modes 1, 2 or 3), an attempt to write to a pixel outside the window causes the V (overflow) bit in the status register to be set to 1; a write (or attempt to write) to a pixel inside the window sets V to 0. When window checking is turned off (mode 0), the V bit is unaffected during pixel writes.

7.10.1 W=1 Mode - Window Hit Detection

The W=1 mode detects attempts to write to pixels within the window. This form of window checking supports applications which permits objects on the screen to be *picked* by pointing to them with a cursor. In this mode, **all** pixel writes are inhibited, whether they address locations inside or outside the window. A window violation interrupt is requested on an attempt to write to a pixel inside the window.

For the PIXBLT and FILL instructions, the V (overflow) bit is set to 1 if the destination array lies completely outside the window. No interrupt request is generated (the WVP bit in the INTPEND register is not affected) in this case. However, if any pixel in the destination array lies within the window, the V bit is set to 0 and a window violation interrupt is requested (the WVP bit is set to 1). If the interrupt is enabled, the saved PC points to the instruction that follows the PIXBLT or FILL that caused the interrupt. If the interrupt is disabled, execution of the next instruction begins.

While no pixel transfers occur during the PIXBLT and FILL instructions executed in this mode, the specified destination array is clipped to lie within the window. In other words, the DADDR and DYDX registers are adjusted to be the starting address, width, and height of the reduced array that is the intersection of the two rectangles represented by the destination array and the window. This function can be adapted to determine the intersection of two arbitrary rectangles on the screen - a calculation that is often performed in windowed graphics systems.

In the case of a DRAV or PIXT instruction, an attempt to write to a pixel outside the window causes the V bit to be set to 1. No interrupt request is generated (the WVP bit is not affected). An attempt to write to a pixel inside the window causes the V bit to be set to 0, and a window violation interrupt request is generated (the WVP bit is set to 1).

At the end of a LINE instruction, the V bit is 0 if any destination pixel processed by the instruction lies within the window; otherwise, V is 1. Attempts to write to pixels outside the window do not cause interrupt requests to be generated (the WVP bit is not affected). An attempt to write to a pixel inside the window causes a window violation interrupt to be requested (the WVP bit is set to 1) and the LINE instruction aborts. If the interrupt is enabled, the PC saved during the interrupt points to the instruction that follows the LINE instruction. If the interrupt is disabled, execution of the next instruction begins.

The W=1 mode can be used to pick an object on the screen by means of the following simple algorithm. An object previously drawn on the screen is picked by moving the cursor to the object's position and selecting it. To determine which object is pointed to, the software first sets the window to a small region surrounding the position of the cursor. The software next steps a second time through the same display list used to draw the current screen until one of the objects causes a window interrupt to occur. This should be the object pointed to by the cursor. If no object causes an interrupt, the pick window can be enlarged and the process repeated until the object is found. If two objects cause interrupts, the size of the pick window can be reduced until only one object causes an interrupt.

7.10.2 W=2 Mode - Window Miss Detection

The W=2 mode permits a PIXBLT or FILL instruction to be aborted if any *pixel* in the destination array lies outside the window. The destination array is written only if the array lies entirely within the window, in which case the V (overflow) bit is set to 0, and no interrupt request is generated (the WVP bit is not affected). If any pixel in the destination array lies outside the window, the V bit is set to 1, and a window violation interrupt is requested (the WVP bit is set to 1).

For the DRAV and PIXT instructions, the destination pixel is drawn only if it lies within the window. In this case, the V bit is set to 0, and no interrupt request is generated (the WVP bit is not affected). If the destination location lies outside the window, the pixel write is inhibited, the V bit is set to 1, and a window violation interrupt is requested (the WVP bit is set to 1).

At the end of a LINE instruction, the V bit is 0 if the last destination pixel processed by the instruction lies within the window; otherwise, V is 1. Attempts to write to pixels inside the window do not cause interrupt requests to be generated (the WVP bit is not affected). An attempt to write to a pixel outside the window causes a window violation interrupt to be requested (the WVP bit is set to 1) and the instruction aborts. If the interrupt is enabled, the PC saved during the interrupt points to the instruction that follows the LINE instruction. If the interrupt is disabled, execution of the next instruction begins.

7.10.3 W=3 Mode - Window Clipping

In the W=3 mode, only writes to pixels within the window are permitted; writes to pixels outside the window are inhibited. No interrupt request is generated for any case.

For a PIXBLT or FILL instruction, only the portion of the destination array lying within the window is drawn. At the start of instruction execution, the specified destination array is automatically preclipped to lie within the window before the first pixel is transferred. Hence, no execution time is lost attempting to write destination pixels which lie outside the window. In the case of a PIXBLT, the source array is preclipped to fit the adjusted dimensions of the destination array before the transfer begins.

During execution of a DRAV or PIXT instruction, a write to a pixel inside the window is permitted, and the V bit is set to 0. An attempted write to a pixel outside the window is inhibited, and the V bit is set to 1.

For the LINE instruction, writes to pixels outside the window are inhibited at drawing time; no preclipping is performed. The value of the V bit at the end of a LINE instruction is determined by whether the last pixel calculated by the instruction fell inside (V=0) or outside (V=1) the window.

7.10.4 Specifying Window Limits

The limits of the current window are specified in the WSTART (window start) and WEND (window end) registers. WSTART specifies the minimum XY coordinates in the window, and WEND specifies the maximum XY coordinates.

As Figure 7-10 shows, WSTART specifies the XY coordinates (X_{start},Y_{start}) at the upper left corner of the window, and WEND specified the XY coordinates (X_{end},Y_{end}) at the bottom right corner of the window. The origin is located in its default position in the top left corner of the screen.

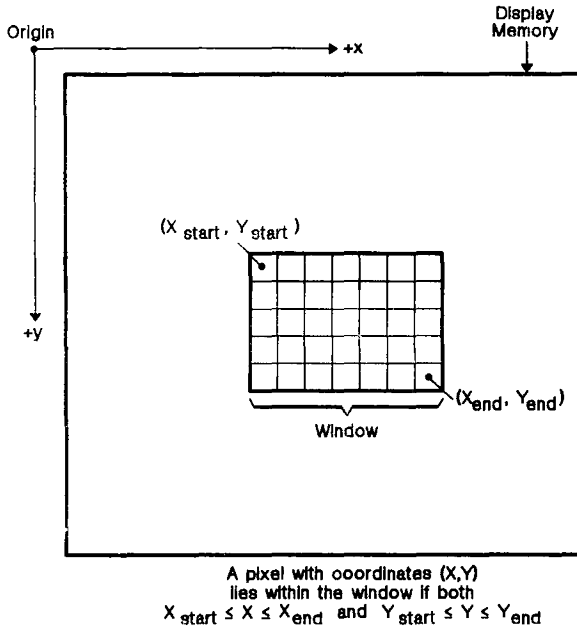


Figure 7-10. Specifying Window Limits

Figure 7-10 shows that a pixel that has coordinates (X,Y) lies within the window if $X_{start} \le X \le X_{end}$ **and** $Y_{start} \le Y \le Y_{end}$. If a pixel does not meet these conditions, it lies outside the window.

When $X_{start} > X_{end}$ **or** $Y_{start} > Y_{end}$, the window is empty; that is, it contains no pixels. Under these conditions, the window checking hardware detects all destination pixel addresses as lying outside the window. Note that the conditions $X_{start} = X_{end}$ **and** $Y_{start} = Y_{end}$ together specify a window containing a single pixel.

Window start and end coordinates must lie in the range (0,0) to (+32767,+32767). A window cannot contain pixels with negative X or Y coordinates.

7.10.5 Window Violation Interrupt

A window violation (WV) interrupt is requested (the WVP bit in the INTPEND register is set to 1) when:

- W=1 and an attempt is made to write to a pixel **inside** the window

  or

- W=2 and an attempt is made to write to a pixel **outside** the window

The interrupt occurs if it is enabled by the following conditions:

- The WVE bit in the INTENB register is 1
- The IE bit in the status register is 1

Alternatively, if the WV interrupt is disabled (IE=0 or WVE=0), the window violation can be detected by testing the value of either the V bit in the status register or the WVP bit following the operation.

When a WV interrupt occurs, the registers that change during the LINE, PIXBLT and FILL instructions contain their intermediate values at the time the violation was detected.

7.10.6 Line Clipping

The TMS34010 supports two methods for clipping straight lines to the boundaries of a rectangular window: postclipping and preclipping. Postclipping means that just before each pixel on the line is drawn, it is compared with the window limits. If it lies outside the window, the write is inhibited. In contrast, preclipping involves determining in advance of any drawing operations which pixels in the line lie within the window. The algorithm draws only these pixels, and makes no attempt to write to pixels outside the window. A preclipped line may take less time to draw since no calculations are performed for pixels lying outside the window. In contrast, postclipping spends the same amount of time calculating the position of a pixel outside the window as it does calculating a pixel inside the window.

When postclipping is used, special window comparison hardware compares the coordinates of the pixel being drawn against all four sides of the window at once. The W=3 window-checking mode is selected, and window checking is performed in parallel with execution of the LINE instruction, so no overhead is added to the time to draw a pixel. However, unless this form of clipping is used carefully, another type of overhead may become significant. For example, in a CAD (computer-aided design) environment where only a small portion of a system diagram is to be displayed at once, potentially a great deal of time could be spent performing calculations for points (or entire lines) lying off-screen.

Preclipping is generally faster than postclipping, depending on how likely a line is to lie outside the window. The first step in preclipping a series of lines is to identify those that lie either entirely inside or outside the window. This is accomplished by using an "outcode" technique similar to that of the Cohen-Sutherland algorithm. Those lines lying entirely outside are "trivially rejected" and consume no more processing time. Those lines lying entirely

within are drawn from one endpoint to the other with no clipping required. This still leaves a third category of lines that may cross a *window boundary*, and these require intersection calculations. However, this technique is powerful for reducing the number of lines that require such calculations. While the calculation of outcodes could be performed in software, this would represent significant overhead for each line considered. The TMS34010 provides a more efficient implementation via its CPW (compare point to window) instruction, which compares a point to all four sides of the window at once.

The outcode technique classifies a line according to where its endpoints fall in relation to the current clipping window. The area surrounding the window is partitioned into eight regions, as indicated in Figure 7-11. Each region is assigned a 4-bit code called an *outcode*. The outcode within the window is 0000_2. When an endpoint of a line falls within a particular region, it is assigned the outcode for that region. If the two endpoints of a line both have outcodes 0000_2, the line lies entirely within the window. If the bitwise AND of the outcodes of the two endpoints yields a value other than 0000_2, the line lies entirely outside the window. Lines that fall into neither of these categories may or may not be partially visible within the window.

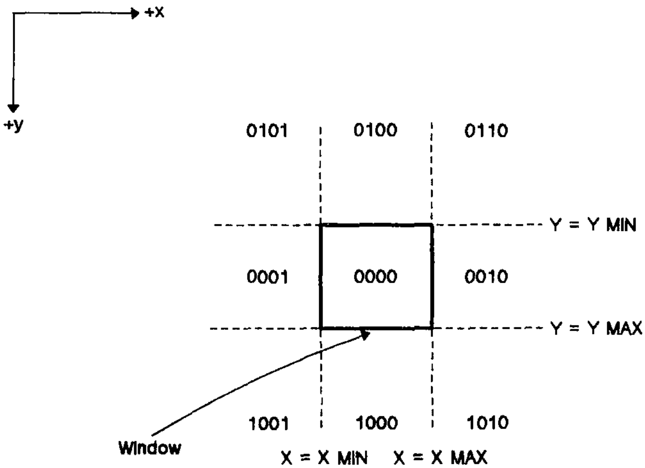
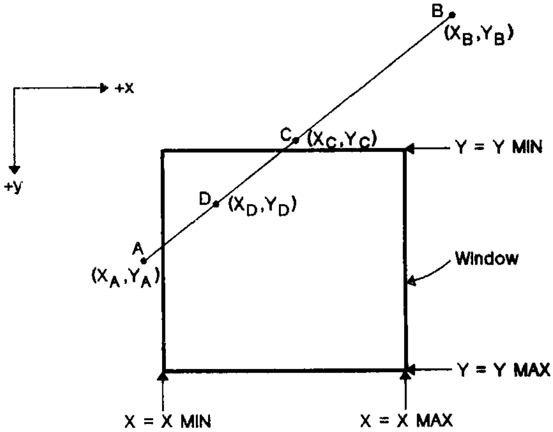


Figure 7-11. Outcodes for Line Endpoints

For those lines that require intersection calculations after the outcodes have been determined, midpoint subdivision is an efficient means of preclipping. The object again is to ensure that drawing calculations are performed only for pixels lying within the window. An example of the midpoint subdivision technique is illustrated in Figure 7-12. The line *AB* lies partially within the window. The first step is to determine the coordinates of the line's midpoint at *C*. These are calculated as follows:

$$(X_C, Y_C) = \left(\frac{X_A + X_B}{2} , \frac{Y_A + Y_B}{2} \right)$$



$$(X_C, Y_C) = \left(\frac{X_A + X_B}{2}, \frac{Y_A + Y_B}{2}\right)$$

$$(X_D, Y_D) = \left(\frac{X_A + X_C}{2}, \frac{Y_A + Y_C}{2}\right)$$

Figure 7-12. Midpoint Subdivision Method

Comparing the outcodes of *B* and *C*, segment *BC* lies entirely outside the window and can be trivially rejected. Segment *AC* still lies partially within the window and will be subdivided again. The coordinates of point *D*, the midpoint of *AC*, are calculated as before. Point *D* is determined to lie within the window. The LINE instruction is now invoked two times, for segments *DC* and *DA*, with *D* selected as the starting point in each case. For each segment the W=2 window-checking mode is selected, but the window violation interrupt is disabled. When each line crosses the window boundary, the window-checking hardware detects this and the LINE instruction aborts. In this way the LINE instruction performs drawing calculations only for portions of *DA* and *DC* lying within the window.